1 <div align="center">**JAVA C++ PROXY OBJECTS**</div>

2

3 **Field of the Invention**

4   The invention relates to programming language interactions, and more specifically

5 to legacy C++ GUI interactions with a Java environment.

6 **Background of the Invention**

7   In many software systems today, the existing or legacy graphical user interfaces

8 ("GUIs") are coded in C++ code. Over the past few years, however, Java® has become

9 the programming language of choice. In order to use Java for providing the functionality

10 of a system, the C++ GUIs must be made useable with Java. One alternative is to rewrite

11 all of the lines of C++ code in Java. Rewriting the C++ code may comprise rewriting

12 many lines of code. In many situations this is impractical. Furthermore, in many

13 situations, it is desirable, practical, and cost-effective to maintain the C++ GUIs, instead

14 of replacing them, and to use them to interface Java objects and methods of the Java

15 functional code.

16   Accordingly, another alternative is to enable the C++ GUIs to make calls directly

17 to the Java objects and methods in a Java Virtual Machine ("JVM") in the Java

18 environment. The Java objects and methods control the functions of the computer

19 systems. In order for the C++ GUIs to make calls directly to the Java objects and

20 methods in the JVM, the C++ GUIs must make Java Native Interface ("JNI") Application

21 Programming Interface ("API") calls across the Java to legacy boundary, as conceptually

22 shown in Figure 1. Likewise, Java data types must be converted to C++ data type, and

23 vice-versa.

24   A solution to this requirement comprises coding JNI APIs and data type

25 conversions into each of the C++ GUIs as required. This is disadvantageous because it

26 litters or clutters the legacy C++ GUIs with the JNI API and data type conversion coding,

27 bloating the size of the C++ GUIs and causing future maintenance problems. Moreover,

28 many of the JNI APIs are repetitious among the C++ GUIs. Therefore, coding the JNI

29 APIs into each of the C++ GUIs is inefficient. Additionally, when the C++ code is

30 converted to Java, the JNI APIs must be deleted, as they will be unnecessary when the

31 former C++ GUIs are on the Java side of the JNI.

32 **Summary of the Invention**

33   The present invention comprises a system and method for enabling the efficient

34 accessing of Java objects and methods by legacy C++ GUIs. A base proxy object

1 according to the present invention encapsulates the JNI APIs necessary for calling the

2 Java methods in order to manipulate the Java objects. The base proxy object also

3 comprises the necessary mapping mechanism for converting Java data types to C++ data

4 types, and vice-versa. The JNI APIs and mapping mechanism are contained in general

5 functions coded into the base proxy object.

6 A method of efficiently accessing Java objects, classes and methods by legacy

7 C++ GUIs according to the present invention comprises a C++ GUI obtaining a Java

8 object, the base proxy object creating a C++ object that proxies the Java object, the C++

9 GUI executing a callback that sends a request, via the C++ proxy object, to the base proxy

10 object. The request comprises the Java object name, a class name, a method name, a

11 C++ data type and, if setting a Java attribute, data of the C++ data type. The base proxy

12 object processes the request and makes the necessary JNI API calls to pass it to the JVM.

13 The base proxy object obtains the Java method ID, calls the Java object, class and

14 method, and gets or sets the Java attribute. If the callback gets an attribute, the base

15 proxy object converts the retrieved Java attribute from the Java data type to the C++ data

16 type and sends the converted attribute to the C++ proxy object. If the callback sets an

17 attribute, the base proxy object converts the data from the C++ data type to the Java data

18 type prior to sending the data through the JNI layer.

19 **Brief Description of the Figures**

20 The detailed description will refer to the following drawings, in which like

21 numbers refer to like items, and in which:

22 Figure 1 is a block diagram conceptually illustrating legacy C++ GUI interaction

23 with Java objects in a Java Virtual Machine.

24 Figure 2 is a block diagram of a computer system on which the present invention

25 may be run.

26 Figure 3 is a block diagram conceptually illustrating a base proxy object according

27 to the present invention.

28 Figure 4a is a static structure diagram of an embodiment of the base proxy object

29 and interaction between C++ GUIs and Java objects according to the present invention.

30 Figure 4b is a static structure diagram of an embodiment of C++ proxy object

31 according to the present invention.

32 Figure 5 is a flowchart of a method for creating a C++ proxy object according to

33 the present invention.

1   Figure 6 is a flowchart of a method of using the base proxy object to facilitate

2 C++ GUI access to Java objects according to the present invention.

3   Figure 7 is a sequence diagram illustrating the use of the base proxy object

4 according to the present invention.

5 **Detailed Description of the Invention**

6   The present invention may be used with computer systems that utilize C++

7 graphical user interfaces ("GUIs") to access Java objects across a Java Native Interface

8 ("JNI"). Figure 2 illustrates a computer network system with which the present invention

9 may be used. The network system 10 comprises a ServiceControl Manager ("SCM") 12

10 running on a Central Management Server ("CMS") 14 and one or more nodes 16

11 managed by the SCM 12 on the CMS 14. Together the one or more nodes 16 managed

12 by the SCM 12 make up a SCM cluster 17. A group of nodes 16 may be organized as a

13 node group 18.

14   The CMS 14 preferably is an HP-UX 11.x server running the SCM 12 software.

15 The CMS 14 includes a memory (not shown), a secondary storage device, a processor, an

16 input device (not shown), a display device (not shown), and an output device (not shown).

17 The memory, a computer readable medium, may include, RAM or similar types of

18 memory, and it may store one or more applications for execution by processor, including

19 the SCM 12 software. The secondary storage device, a computer readable medium, may

20 include a hard disk drive, floppy disk drive, CD-ROM drive, or other types of non-

21 volatile data storage. The processor executes the SCM 12 software and other

22 application(s), which are stored in memory or secondary storage, or received from the

23 Internet or other network 24, in order to provide the functions and perform the methods

24 described in this specification, and the processing may be implemented in software, such

25 as software modules, for execution by the CMS 14 and modes 16. The SCM 12 is

26 preferably programmed in Java® and operates in a Java® environment that is preferably

27 accessed by using legacy C++ GUIs and the present invention. See <u>ServiceControl</u>

28 <u>Manager Technical Reference, HP® part number: B8339-90019</u>, available from Hewlett-

29 Packard Company, Palo Alto, CA., which is hereby incorporated by reference, for a more

30 detailed description of the SCM 12. The <u>ServiceControl Manager Technical Reference,</u>

31 <u>HP® part number: B8339-90019</u> is also accessible at

32 <u>http://www.software.hp.com/products/scmgr</u>.

33   Generally, the SCM 12 supports managing a single SCM cluster 17 from a single

34 CMS 14. All tasks performed on the SCM cluster 17 are initiated on the CMS 14 either

1   directly or remotely, for example, by reaching the CMS 14 via a web connection 20.

2   Therefore, a workstation 22 at which a user sits only needs a web connection 20 over a

3   network 24 to the CMS 14 in order to perform tasks on the SCM cluster 17. The

4   workstation 22 preferably comprises a display, a memory, a processor, a secondary

5   storage, an input device and an output device. In addition to the SCM 12 software and

6   the HP-UX server described above, the CMS 14 preferably also comprises a data

7   repository 26 for the SCM cluster 17, a web server 28 that allows web access to the SCM

8   12 and a depot 30 comprising products used in the configuring of nodes, and a I/UX

9   server 32.

10   The nodes 16 are preferably HP-UX servers or other servers. The nodes 16 may

11   be referred to as "*managed nodes*" or simply as "*nodes*". Conceptually, the node 16

12   represents a single instance of HP-UX running on some hardware. The node 16 may

13   comprise a memory, a secondary storage device, a processor, an input device, a display

14   device, and an output device.

15   Although the CMS 14 is depicted with various components, one skilled in the art

16   will appreciate that the CMS 14 may contain additional or different components. In

17   addition, although aspects of an implementation consistent with the present invention are

18   described as being stored in memory, one skilled in the art will appreciate that these

19   aspects can also be stored on or read from other types of computer program products or

20   computer-readable media, such as secondary storage devices, including hard disks, floppy

21   disks, or CD-ROM; a carrier wave from the Internet or other network; or other forms of

22   RAM or ROM. The computer-readable media may include instructions for controlling

23   the CMS 14 (and/or the nodes 16) to perform a particular method, such as those described

24   herein.

25   Java objects operating in a JVM provide the functionality of the SCM 12. In the

26   system 10, when a user, through a C++ GUI, wants to access the functionality of the SCM

27   12 (*e.g.*, to create, retrieve, save, delete or modify persistent data (*e.g.*, in the data

28   repository 26) of the Java objects or to run a tool on a node(s) or node group(s)), the C++

29   GUI executes a callback that causes a particular Java class to be instantiated, thus creating

30   a particular Java object and a C++ object, that proxies the Java object. Java classes are

31   meta-definitions that define the structure of a Java object. Java classes when instantiated

32   create instances of the Java classes and are then considered Java objects. Methods within

33   Java objects are used to get or set attributes of the Java object and to change the state of

34   the Java object. Consequently, the C++ GUI may subsequently execute callbacks that

1   execute methods on the proxy C++ object and that are processed by a base proxy object

2   of the present invention to run the methods of the Java object to get or set attributes of the

3   Java object, thereby creating, retrieving, saving, deleting or modifying data in the

4   persistent store or running a tool on a node(s) or node group(s).

5   Some of the objects and classes discussed herein are named with a prefix "mx".

6   The mx prefix is indicative of the application utilizing the objects and classes (e.g., the

7   SCM 12) and is merely exemplary. Indeed, the names of classes, objects and methods

8   discussed herein are exemplary, are not intended to be limiting, and are merely used for

9   ease of discussion.

10   Figure 3 conceptually illustrates a base proxy object 40 and interactions between a

11   C++ environment 42 and a JVM 48. For clearer understanding, the reference numbers

12   used in Figure 3 are used throughout this specification. In the present embodiment, a

13   base proxy object 40 is coded in the SCM 12 software and stored in the CMS 14. The

14   base proxy object 40 is shown in a C++ environment 42, with one or more C++ GUIs 43

15   and one or more C++ proxy objects 44, separated by a JNI boundary 46 from the JVM 48

16   and one or more Java objects 50 that each comprise one or more methods.

17   The C++ proxy objects 44 proxy the Java objects 50 and each C++ proxy object

18   44 is created when a C++ GUI 43 obtains a Java object 50, as described below.

19   Accordingly, each C++ proxy object 44 includes methods that correspond to the methods

20   of the Java object 50 that the C++ proxy object 44 proxies. The C++ proxy objects 44

21   also maintain names of the Java objects 50 that they proxy, identifying the Java classes

22   that need to be instantiated to create instances of the Java objects 50. As is discussed

23   below, when the C++ GUI 43 sends a request to a C++ proxy object 44 (i.e., executes a

24   callback that executes a method on the C++ proxy object 44), the C++ proxy object 44

25   passes the method name, corresponding to the Java object 50 method name to the base

26   proxy object 40. Each instance of a C++ proxy object 44, and the Java object 50 that the

27   C++ proxy object 44 proxies, will exist for the duration of execution of the C++ GUI 43

28   that obtained the Java object 50.

29   Referring again to Figure 3, the base proxy object 40 hides the mechanics of using

30   the JNI from C++ consumers (GUIs 43) of Java objects. By hiding these mechanics, the

31   base proxy object 40 makes it appear to C++ GUIs 43 that the C++ GUIs 43 are

32   interacting with and manipulating fully functional C++ objects 44 when the C++ GUIs 43

33   execute callbacks to get or set attributes of the Java objects 50 that the C++ objects 44

34   proxy.

1   The base proxy object 40 is a base C++ class that provides a basic mapping

2   mechanism. The C++ proxy objects 44 are sub-classes of the base proxy object 40. The

3   base proxy object 40 preferably caches both Java Class IDs as well as Java Method IDs in

4   order to minimize the number of C++ to Java Virtual Machine ("JVM") transitions. The

5   base proxy object 40 preferably uses the C++ Standard Template Library (STL) to

6   implement both the caching and "method name (Java) to method ID (Java)" mapping

7   mechanisms.

8   As described above, a JNI API call is required for a C++ GUI 43 to access a Java

9   object 50 across the JNI boundary 46. The base proxy object 40 preferably includes the

10   JNI API calls necessary to access Java objects 50 requested by all of the C++ GUIs 43.

11   The JNI API calls are coded into general functions (functions and methods are used

12   interchangeably herein with regards to C++ objects) in the base proxy object 40 that call a

13   Java method to either get or set an attribute of a specific type in the Java object being

14   accessed (for exemplary general functions, see Figure 4a). Coding the JNI API calls

15   necessary to access Java objects 50 requested by all of the C++ GUIs 43 in the base proxy

16   object 40 collects these JNI API calls in one class, instead of littering or cluttering each

17   C++ GUI 43 with the JNI API calls necessary for that individual C++ GUI 43.

18   Collecting the JNI API calls in the base proxy object 40 has a number of

19   advantages: for example, since some JNI API calls would otherwise be used by more than

20   one C++ GUIs 43, collecting them in the base proxy object 40 prevents unnecessary

21   repetition, allows generalization (*e.g.*, getting a user name is generalized as getting a

22   String with a specific method call) and increases efficiency; and, the C++ GUIs 43 are

23   kept streamlined, which is especially advantageous if the legacy C++ GUIs 43 are ever

24   converted to Java. Conversion of the legacy C++ to Java may basically comprise the re-

25   writing of C++ syntax to Java syntax; by keeping the JNI API calls out of the C++ GUIs

26   43 and in the base proxy object 40, the base proxy object 40 and C++ proxy objects 44

27   may simply be removed when the legacy C++ is converted to Java. These and other

28   advantages will become apparent throughout this specification.

1   Java data types and C++ data types are different. Therefore, when accessing the

2   JVM 48 through the C++ environment 42, as illustrated by Figure 3, it is necessary to

3   map the Java data types to C++ data types. Accordingly, the base proxy object 40

4   comprises a mapping between Java data types and C++ data types. The mapping

5   preferably used is shown in the following table:

Java Data Type	C++ Data Type
String	String
boolean (Bool)	bool
long	long long
int	long
Array of int	vector of long
Array of object	vector of proxy object
Array of String	vector of string

15   Table 1

16   The conversion according to this mapping is preferably coded into the base proxy object

17   40 general functions or methods, described above, that call a Java method to get or set an

18   attribute of a specific type in the Java object being accessed.

19   Referring back to Figure 3, the general interaction between the C++ GUIs 43, C++

20   proxy objects 44, the base proxy object 40 and the Java objects 50 is shown. Assuming

21   that a C++ proxy object 44 proxying a Java object 50 has already been created (as

22   described below), a C++ GUI 43 executes a callback that in turn executes a method on the

23   C++ proxy object 44 (corresponding to a method of the proxied Java object 50) to get or

24   set an attribute(s) in the proxied Java object 50. The C++ proxy object 44 passes control

25   through the GUI callback to the base proxy object 40 as a method call with arguments

26   including a Java object name, a class name, a method name(s), and the C++ data type of

27   the output the C++ object 40 is getting or the input the C++ object 40 is setting. The Java

28   object name, class name and method name(s) passed by the C++ proxy object 44 to the

29   base proxy object 40 identify the proxied Java object 50, the Java class that defines the

30   structure of the proxied Java object 50, and a method(s) of the Java object 50 that the C++

31   GUI 43 needs to access. The base proxy object 40 processes the method arguments to

32   access the Java object and to call the requested method(s) across the JNI boundary 46.

33   In order to call a Java method across the JNI boundary 46, a method ID for the

34   Java method is required. The method ID is a unique identifier associated with a Java

1   method that is established when the Java class containing the Java method is initialized.

2   The method ID retains its value until the Java class containing the method ID is unloaded.

3   The base proxy object 40 obtains the method ID across the JNI boundary 46 using the

4   method name provided by the C++ proxy object. When a C++ proxy object 44 is first

5   created (as described below) the method IDs and class ID are preferably dynamically

6   obtained as instance data from the Java objects 50 and cached. When the method is

7   subsequently called, the method ID may be accessed from the cache without having to

8   access the method ID via the JNI, thereby reducing the number of C++ to JVM

9   transitions. The cache may be a method data hash table, wherein the method data

10   comprises the method signature and the method ID.

11   Accordingly, the base proxy object 40 preferably processes the method request by

12   identifying the Java object 50 using the Java object name provided by the C++ proxy

13   object 44, obtaining the method ID from the cache, and calling the Java object 50 and

14   Java method(s) via the JNI 46, using the object name and method ID. If a set function is

15   called, the base proxy object 40 will pass the input data supplied by the C++ GUI 43 to

16   the Java object 50.

17   If the executed callback specifies a get function (*i.e.*, returning an output from the

18   Java object), the called Java method will return a Java data type output and the base proxy

19   object 40 will convert the output to the C++ data type specified by the above mapping in

20   Table 1. If the executed callback specifies a set function (*i.e.*, sending input data to the

21   Java object), the base proxy object 40 will convert the input data from a C++ data type

22   supplied by the callback to the Java data type specified by the above mapping in Table 1

23   before sending the input data to the Java object 50.

24   Figure 4a is a static structure diagram that illustrates an exemplary embodiment of

25   the base proxy object 40. The base proxy object 40 is a base implementation class

26   labeled 'MxProxy' in Figure 4a. The base proxy object 40 comprises the following data

27   variables 401: myJNIEnv, a JNI environment pointer obtained from an Object Action

28   Manager ("ObAM") (alternatively, the JNI environment pointer may be obtained directly

29   from JNI API calls); myJavaObject, a Java object reference to the Java object proxied by

30   the C++ object 44 currently sub-classing the base proxy object 40; myJavaClass, a Java

31   class reference to the proxied Java class; myClassName, the name of the proxied Java

32   class; and, myMethodData, for accessing the method data hash table (containing the

33   method IDs and signatures) for the proxied Java class. The Java object reference and the

1   name of the proxied Java class are preferably provided by the C++ GUI 43 when it

2   instantiates the C++ proxy object 44.

3   The base proxy object 40 also comprises base proxy object functions or methods

4   402 and the general functions or methods 403 mentioned above. The parameters of the

5   functions 402 and 403 are shown within the parenthesis and the returned data or data type

6   is shown after the colon. The base proxy object functions 402 include a constructor (init),

7   a copy constructor (MxProxy), and getJavaObject, get ClassName, getJniEnv and

8   getMethodID functions. The constructor is used to initialize the C++ proxy objects and is

9   called by the C++ object constructors called below. The copy constructor is used to copy

10   the C++ proxy objects if the C++ proxy objects are stored in vectors. The getJavaObject

11   function returns the proxied Java object reference from the myJavaObject variable 401 so

12   that it may be passed to the JVM. The getClassName function is used to return the Java

13   class name from the myJavaClass variable 401. The getJniEnv function returns the JNI

14   environment pointer returned from a call to ObamGetJNIEnv() or an appropriate JNI API

15   call; the current JNI environment pointer is retrieved in order to make a call to the JNI

16   API. The getMethodID provides a protected method for acquiring a method ID from an

17   instantiated class (*e.g.*, the proxied Java class).

18   The general functions 403 call a Java method to get or set an attribute of a specific

19   type in the Java object being accessed. The inventors realized that getting or setting

20   attributes of any Java object generally meant getting or setting a data of a particular data

21   type by running a particular method. For example, the function getString(in

22   methodName: string &): string *, will return a pointer to a C++ STL string using the

23   method identified by methodName, which is a string provided by the C++ proxy object

24   44, and return a pointer to a C++ string. As an example, assuming a C++ proxy object 44

25   MxUser with a method name called getUserName(): string *, a C++ GUI callback method

26   invocation comprising the name call getUserName(): string * will be executed as a

27   getString with getUserName as the methodName. The getString(getUserName) will

28   retrieve a String from the Java method getUserName and convert the String to a C++

29   string for output to the C++ proxy object 44 and to the C++ GUI 43.

30   Figure 4a also illustrates some of the C++ proxy objects 44 that are sub-classes

31   from the base proxy object. The C++ proxy objects 44 are not created until a C++ GUI

32   43 (not shown in Figure 4a) obtains a Java object 50 (not shown in Figure 4a). However,

33   the constructor functions (*i.e.*, Mx{object}) that create the C++ proxy objects 44 and the

34   methods (*e.g.*, +setName(in toolName: string &)) that the C++ proxy objects 44 comprise

1   are pre-coded *(e.g.,* as part of the SCM 12 software). For example, MxTool is a C++

2   proxy object 44 that proxies a Java object 50 of the same name. MxTool is created when

3   a C++ GUI 43 obtained an empty Java object 50 MxTool according to the process

4   described below. For each instance of Java object 50 MxTool, there will be a

5   corresponding C++ proxy object 44 MxTool. Consequently, each time a C++ GUI 43

6   creates a new tool or wishes to modify *(e.g.,* change or delete) an existing tool, a Java

7   object 50 MxTool and C++ proxy object 44 MxTool will be created and exist for the

8   duration of execution of the C++ GUI 43.

9   In a preferred embodiment, a plurality of Java objects 50 provide functionality for

10   a computer system. Related to the Java objects 50 are a corresponding number of C++

11   proxy objects 44 that proxy the Java objects 50. In the computer network system 10

12   illustrated in Figure 2, the SCM 12 includes the following C++ proxy objects 44 proxying

13   Java objects 50 with the same name: MxUser, MxUserName, MxNode, MxNodeName,

14   MxIPAddress, MxNodeGroup, MxNodeGroupName, MxTool, MxToolName,

15   MxToolGroupName, MxToolToolGroupPair, MxFileCopyPair, MxParameter,

16   MxRunnableTool, MxTaskStatus, MxTargetStatus, MxTargetOutput, MxAuthorization,

17   MxRole, MxRoleName, MxString. Consequently, C++ GUIs 43 may access all of the

18   Java objects 50 with the above names, through the corresponding C++ proxy objects 44

19   and the base proxy object 40.

20   A method data hash table 404 *(e.g.,* MxMethNameToMethData) is also shown in

21   Figure 4a. MxMethNametoMethData is an object that is a data member in the C++ base

22   proxy class 40. Initially, MxMethNametoMethData is an empty hash table 404. Each

23   C++ proxy object 44, such as MxNode, preferably has a method info array *(e.g.,* an array

24   methodInfo_t object) that is a subset of data that goes into the

25   MxMethodNametoMethData hash table 404. The methodInfo_t data preferably contains

26   the method name and signature. Preferably, the methodInfo_t data does not include the

27   method ID, since this is obtained dynamically, as described herein. The methodInfo_t

28   data is placed inside the hash table 404 in order to provide fast and efficient access to the

29   methodInfo_t data.

30   As implied by the name MxMethNameToMethData, the method name is the key

31   to the hash table 404 and the methodInfo_t data is retrieved from the hash table 404

32   simply by using the method name. If the methodInfo_t data were left as an array, this

33   array would have to be linearly searched to locate the correct data. The methodInfo_t

34   data is entered into the hash table 404 by passing the methodInfo_t object to the

1   MxMethNameToMethData object and asking the MxMethNameToMethData object to

2   populate itself with the passed methodInfo_t object.

3   Figure 4b illustrates an exemplary C++ proxy object 44, MxTool, including

4   MxTool's constructor function and get/set functions. Other C++ proxy objects 44

5   likewise comprise a constructor function and get/set functions. As shown in the static

6   structure diagram in Figure 4b, MxTool comprises methods 441 for creating and

7   modifying tools that are used in the network system 10 and with the SCM 12 described

8   above. To create a tool, a C++ GUI 43 issues a request with the appropriate user-entered

9   data (*e.g.*, tool name, created by, created time (*e.g.*, as determined by the network system

10   10), description, etc.) as the method parameters (in the parenthesis) to the MxTool C++

11   proxy object 44. The executed C++ GUI 43 callback makes the necessary method calls

12   (*e.g.*, *setName(in toolName: string &), setCreatedBy(in creatorID: long), set*

13   *CreatedTime(in milliseconds: long long), setDescription(in description: string&)*, etc.),

14   on the MxTool C++ proxy object 44 that have the effect of calling the proxied Java object

15   50 through the base proxy object 40. The base proxy class 40 processes the method

16   request(s) and makes the necessary JNI API calls, sending the appropriate Java

17   methodIDs and making the appropriate data type conversions. Other C++ GUI 43

18   method calls (requests) to the other C++ proxy objects 44 work in a similar manner.

19   Another advantage of the base proxy object 40 of the present invention is that it is

20   customizable. Since it is customizable, the base proxy object 40 may be used in any

21   system in which a C++ to Java transition similar to that described above takes place. The

22   base proxy object 40 may be customized by adding or removing base proxy object

23   functions 402 or general functions 403. For example, the base proxy object 40 shown in

24   Figure 4a does not include getting or setting of floats. If a particular system required the

25   getting or setting of floats, getFloat and/or setFloat functions could be added to the base

26   proxy object 40. Likewise, additional or different the C++ proxy objects 44 may be

27   implemented to mirror additional or different Java objects 50, as determined by the

28   particular needs of the computer system (*e.g.*, network system 10) with which the

29   invention is used.

30   Figure 5 illustrates a method 60 for creating a C++ proxy object 44 comprising a

31   C++ GUI 43 obtaining a Java object 62, initiating C++ proxy object linkage to the Java

32   object 64 and passing instance data to the base proxy object 66. The obtaining step 62

33   may comprise instantiating a Java class to create a Java object 50 in the JVM 48. The

34   C++ GUI 43 may accomplish this via a JNI call that returns a Java object (*e.g.*,

1   GetObjectArrayElement) or via a call to an Objectifier. The Objectifier is a Java object

2   that is a mapping layer that maps C++ procedural code to Java object-oriented classes and

3   that creates new Java objects based on procedural calls from C++ GUIs 43, among other

4   things.

5   When the new Java object is obtained, the JNI environment pointer, Java object

6   name and Java class name are returned as a call to the C++ environment 42, therefore

7   executing a constructor function to create and link a C++ proxy object 44 to the Java

8   object 50. Accordingly, the initiating C++ proxy object linkage to the Java object 64 may

9   comprise calling the appropriate constructor function with the JNI environment pointer,

10   Java object name and Java class name as parameters to create and link a C++ proxy object

11   44 to the new Java object 50. For example, a call comprising the Java object name

12   MxUser will execute a constructor that creates a C++ proxy object 44 of the same name

13   (MxUser). The new C++ object MxUser will be linked, and therefore will proxy, the Java

14   object MxUser.

15   When created, the new C++ proxy object 44 calls a constructor in the base proxy

16   class 40 to initialize the instance data and to create global references for the proxied Java

17   object 50. Therefore, the passing instance data to the base proxy object 66 may comprise

18   the new C++ proxy object 44 calling the init function in the base proxy object and passing

19   the JNI environment pointer, the Java object name and the Java class name with the init

20   call (see Figure 4a). The method 60 may also comprise a step of populating a method

21   name to method date hash table 68. The hash table links method names with method data

22   (*e.g.*, method signature). The populating step 68 preferably comprises the new C++

23   proxy object 44 passing method names and method data to the hash table 404. The

24   method names and method data are instance data of the new C++ proxy object 44. As

25   noted above, maintaining the method data in the hash table 405 enables easier access to

26   the method data by the base proxy object 40 and reduces the number of JNI API calls

27   necessary.

28   After a Java object 50 and its proxy, the C++ object 44, have been created, the

29   C++ GUI 43 can get/set attributes in the Java object 50, through the C++ proxy object 44

30   and the base proxy object 40. Figure 6 illustrates a method 70 for getting/setting

31   attributes in the Java object 50. The method 70 corresponds to the general interaction

32   between the C++ GUIs 43, C++ proxy objects 44, the base proxy object 40 and the Java

33   objects 50 shown in Figure 3 and described above. The method 70 comprises a C++ GUI

34   issuing a request to a C++ proxy object 72, the C++ proxy object passing method data to

1   the base proxy object 74, the base proxy object processing the method data 76 and the

2   Java object executing a Java method 78. The method steps may be performed as

3   described above with reference to Figures 3, 4a and 4b and as illustrated by the exemplary

4   sequence diagram described below.

5   Accordingly, issuing a request to a C++ object 72 preferably comprises the C++

6   GUI 43 invoking a method call on the C++ proxy object 44 and providing the user

7   entered data as the necessary parameters of the invoked C++ proxy object 44 method.

8   The invoked C++ proxy object 44 method corresponds to the Java object 50 method that

9   needs to be invoked to get or set attributes of the proxied Java object 50.

10   Passing method data to the base proxy object 74 preferably comprises the C++

11   proxy object 44 processing the method call invoked by the C++ GUI 43 and calling a base

12   proxy object 40 method. The C++ proxy object 44 includes the method data (*e.g.*, Java

13   method name and the user entered data) in the base proxy object 40 method call. The

14   C++ proxy object 44 method invoked by the C++ GUI 43 determines the base proxy

15   object 40 method called. For example, a common C++ proxy object 44 method,

16   setCreatedBy(const long uid):void, invoked by the C++ GUI 43 to set the uid of a created

17   Node, User, Tool, etc., will call the base proxy object 40 method setInt(const string &

18   methodName, const long cppInt): void when processed, since a uid has a C++ data type

19   long that maps to Java data type Int (as seen in Table 1 above). Consequently, setting the

20   Creating By attribute is basically setting a Java int.

21   Processing the method data 76 preferably comprises the base proxy object 40

22   executing the called method, getting the Java method ID using the method name provided

23   by the C++ proxy object 44, issuing necessary JNI API calls with the method ID to call

24   the Java method indicated by the method ID, and converting C++ data to Java data (and

25   vice-versa). Executing a Java method 78 preferably comprises a Java object 50, which is

26   proxied by the C++ object 44, executing the Java method called by the base proxy object

27   40 and identified by the method ID. If the Java method is a get method, the Java object

28   50 returns a pointer to the C++ data 79.

29   Figure 7 is a sequence diagram detailing an exemplary process according to the

30   present invention utilizing the base proxy object 40 to get and set attributes of a Java

31   object 50 via a C++ GUI 43. The sequence diagram illustrates boxes representing an

32   MxNewNode (a C++ GUI 43) 90, an MxObjectifier 92 (the Objectifier described above),

33   a MxNode (a C++ proxy object 44) 94, a MxProxy (the base proxy object 40) 96, a JNI

34   (the JNI boundary 46) 98 and a MxMethNameToMethData 100 (the method data hash

1   table 405 discussed above). The representative boxes 90-100 are shown with vertical

2   time-lines 102 running from the boxes 90-100 and horizontal call lines 104 running from

3   the appropriate initiating GUI, object, etc. time-line 102 to the appropriate target object,

4   layer, map, etc. time-line 102. The horizontal call lines include a method, with the

5   method parameters within parenthesis and the returned data following a colon. The

6   vertical position of the call lines 104 indicates their sequential order of initialization and

7   execution. The length of the vertical time-line 102 indicates the period of execution of

8   the GUI, object, layer, map, etc. represented by the box 90-100 from which each time-line

9   102 descends. Also shown are notation boxes 106 that comprise various comments. The

10   sequence diagram of Figure 7 is described below moving left to right and top to bottom.

11   In the exemplary process shown, the MxNewNode box 90 represents a C++ GUI

12   43 that may be used to enter data to create a new node in the network system 10. As seen

13   in Figure 7, the MxNewNode GUI 43 issues a call (shown by the getNode():jobject call

14   line 104) to obtain a Java object 50. As noted above, a C++ GUI 43 may obtain a Java

15   object 50, for example, via a JNI call that returns a Java object or via a call to the

16   Objectifier. In the present example, as noted by the associated notation 106, the

17   MxNewNode GUI obtains the Java object 50 via a call to the Objectifier (represented by

18   MxObjectifier box 92). A new, empty MxNode Java object 50 is thus created in the JVM

19   48.

20   The Objectifier returns the new, empty MxNode Java object 50 as a call to the

21   C++ environment 42. This call is illustrated by the MxNode(JNIEnv * env,jobject

22   javaObject,const string & className) call line 104 extending from the MxNewNode 90

23   vertical time-line 102 to the MxNode 94 time-line 102. If the Java object 50 were

24   returned by a JNI call, or in another manner, the Java object 50 would also be returned to

25   the C++ environment 42 as a call. As noted by the associated notation 106, the MxNode

26   call initiates C++ proxy object 44 linkage by calling the appropriate constructor function

27   (i.e., MxNode) in the C++ environment 42. With the parameters provided within the

28   parenthesis, the MxNode call links the new MxNode C++ proxy object 44 to the new

29   MxNode Java object 50.

30   The MxNode box 94 represents the new MxNode C++ proxy object 44 created by

31   the MxNode call. The new MxNode C++ proxy object 44 issues a call to the base proxy

32   class 40 invoking a function in the base proxy class 40 (i.e., the init function) to initialize

33   the instance data and to create global references for the proxied Java object 50. This

34   function call is shown by the init(JNIEnv * env, jobject javaObject, const string &

1   className): void call line 104. The ":void" indicates that the method call returns a void

2   (*i.e.*, nothing). The env is the JNI Environment pointer obtained from ObAM or directly

3   via a JNI API call (as discussed above), the javaObject reference is the Java object

4   reference obtained by the GUI (*e.g.*, from the Objectifier) and the className is the name

5   of the proxied Java class also provided by the GUI.

6   As noted above, the init function initializes the instance data and creates global

7   references for the proxied Java object 50 and class. Accordingly, the base proxy object

8   40, represented by the MxProxy box 96, issues appropriate JNI API calls that are coded

9   into the init function. These JNI API calls are illustrated by the

10   NewGlobalRef(jobject):jobject, FindClass(string):jclass, and NewGlobalRef(jclass):

11   jclass call lines 104. The NewGlobalRef call passes the jobject reference included in the

12   init function call. The FindClass call passes the className string included in the init

13   function call. The NewGlobalRef call passes the jclass reference obtained by the

14   FindClass call.

15   Once the instance data is initialized and global references are created, the new

16   MxNode C++ proxy object 44 issues a call to populate the method data hash table 405

17   (represented by the MxMethNametoMethData box 100) with the method data from the

18   proxied Java class. The method data includes a count (methodCount) of the number of

19   methods in the methodInfo_t array and a pointer (methodInfo_t*) to the methodInfo_t

20   array. The methodInfo_t array includes the method name and signatures. The C++

21   proxy object 44 issues this call since the method data , method names, and method

22   signatures are embedded as instance data in the C++ proxy object 44 and the method data

23   hash table 405 is preferably visible to all base proxy object 40 sub-classes. This call is

24   illustrated by the populate(const int methodCount, const methodInfo_t* const &

25   methodInfo):void call line 104. As noted above, the method name is used as a key to

26   access the method data populated in the hash table 405.

27   After the above steps are executed, the C++ GUI 43 may issue a method execution

28   · request to set the name of the Java object 50, as shown in Figure 7. This request is

29   illustrated by the setName(const string & nodeName): void call line 104 extending from

30   the MxNewNode vertical time-line 102. The setName request is passed to the MxNode

31   C++ object 44. The string in the request is the name of the MxNode Java object 50

32   created above that is to be set.

33   The MxNode C++ proxy object 44 passes the request to the base proxy object 40.

34   The setString method of the base proxy object (see Figure 4a above) is preferably used to

1   set the name of the Java object 50, since the setString method makes calls on the proxied

2   Java object 50. To set the name of the node, an MxNodeName object 50 is preferably

3   instantiated. Therefore, the method data (*i.e.*, method name and Java class ID for the

4   MxNodeName Java object 50) is passed to the base proxy object 40 via a setNameObject

5   call, shown by the setNameObject(const string & methodName,const string &

6   classPath,const string & name): void call line 104. The setNameObject call causes the

7   setNameObject method in the base proxy object 40 to be executed.

8   Consequently, the setNameObject method comprises getting a method ID for the

9   method of the MxNode Java object 50 that sets the name attribute of the MxNode Java

10 object 50 and creating a new Java name object from the string specified in the C++ GUI

11 setName method request by calling the Java method that sets the name attribute with the

12 new Java name object.

13   The getMethodID and GetMethodID call lines 104 show the getting a method ID

14 step. The first time getting a methodID, the base proxy object 40 gets the method ID by

15 crossing the JNI boundary 46 using the proxied java class ID, the methodName string and

16 the method signature, all provided by the C++ proxy object 44. Subsequently, the

17 methodID is cached in the method data hash table, from which it may be retrieved as

18 needed.

19   Referring to Figure 7, the base proxy object 40 first calls its own getMethodID

20 method (as indicated by the self-referential getMethodID call line 104 that indicates that

21 MxProxy is the method initiator and target). The invoked getMethodID(string &

22 methodName): jmethodID function calls the hash table 405 (the

23 MxMethNameToMethData object) using the method name as a key. This call retrieves a

24 methodInfo_t object linked to the method name. The "getMethodID" function looks

25 inside the methodInfo_t object and if it finds a non-null java method ID, then the method

26 ID has already been obtained and the getMethodID function does not need to make the

27 call to the JNI 46. The "cached" method ID is simply returned. If however, the java

28 method ID is null, the getMethodID function makes the call to the JNI 46 and caches the

29 returned ID in the methodInfo_t object, and then puts the methodInfo_t object back into

30 the hash table 405 (the MxMethNameToMethData object) for later use if so requested.

31 Then the getMethodID function returns the method ID to the caller.

32   Consequently, Figure 7 illustrates the lookup (lookup(string

33 key,javaMethodData_t & retValue) : bool), the JNI call (GetMethodID(jclass clazz,const

34 char * methodName,const char * signature) : jmethodID) and the put back of the

1   methodInfo_t object (putBack(string key,javaMethodDate_t & refValue) : bool) since the

2   method ID has not been previously retrieved and cached.

3   The newNameObject(const string & javaClassPath, const string & name):jobject

4   is a self-referential method that calls the constructor of a Java name object (*e.g.*,

5   MxNodeName) that takes the Node name string provided by the C++ GUI 43 and returns

6   a Java object global reference, jobject, to the constructed Java name object. The

7   newNameObject method converts the C++ string provided by the C++ GUI 43 to a Java

8   string, finds the method ID of the Java name object constructor, invokes the constructor to

9   create the Java name object and creates a global reference to the new Java name object

10   (not shown in Figure 7). The CallVoidMethod calls the Java method that sets the name of

11   the MxNode Java object 50. As shown, the parameters include the global reference of the

12   MxNode Java object 50 (*i.e.*, the first jobject), the Java method ID, (*i.e.*, the jmethodID),

13   and the global reference of the Java name object MxNodeName (*i.e.*, the second jobject).

14   The CallVoidMethod is the JNI API call to the JVM, in which the method identified by

15   the jmethodID (*i.e.*, the method of the MxNode Java object 50 that sets the name attribute

16   of the MxNode Java object 50) is called. Once the name of the MxNode Java object 50

17   has been set, the global reference of the Java name object MxNodeName is no longer

18   needed. Accordingly, the base proxy object issues a DeleteGlobalRef(jobject) call to

19   delete the jobject global reference of the Java name object MxNodeName.

20   Once the Java object 50 and its proxy, the C++ object 44, have been created, the

21   C++ GUI may get/set attributes of the Java object 50. Again referring to Figure 7, the

22   MxNewNode GUI 90 issues a method call to set the created by attribute of the Java object

23   50. This method call is shown by the setCreatedBy(const long uid):void call line 104.

24   Therefore, this calls the setCreatedBy function in the MxNode C++ object 44, passing a

25   C++ long user id (uid).

26   Since a C++ long is mapped to a Java int, the MxNode C++ proxy object 44

27   passes the parameter to the base proxy object 40 by calling a setInt function. This is

28   shown by the setInt(const string & methodName, const long cppInt): void call line 104.

29   As seen, the setInt call includes a string for the method name and the C++ long for the

30   Java Int as parameters. In this example, the method name is MxNode.setCreatedBy (*i.e.*,

31   the MxNode Java object 50 method that sets the uid (userid) identifying the user that

32   created the node object).

33   When received by the base proxy object 40, the setInt method gets the method ID

34   of the method in the MxNode Java object 50 that sets the setCreatedBy attribute. Once

1   the method ID (jmethodID) is returned (by the self-referential getMethodID method call

2   shown), the base proxy object calls the method, passing the method ID and the long uid

3   (converted to a java Int jint) to the MxNode Java object 50. This is illustrated by the

4   CallVoidMethod(jobject jmethodID jint) call to the JNI. The jobject parameter is the

5   global reference to the MxNode Java object 50.

6   While the invention has been described with reference to the exemplary

7   embodiments thereof, those skilled in the art will be able to make various modifications

8   to the described embodiments of the invention without departing from the true spirit and

9   scope of the invention. The terms and descriptions used herein are set forth by way of

10 illustration only and are not meant as limitations. Those skilled in the art will recognize

11 that these and other variations are possible within the spirit and scope of the invention as

12 defined in the following claims and their equivalents.